# Generating Safe Boundary APIs between Typed EDSLs and Their Environments

Bob Reynders Dominique Devriese Frank Piessens

iMinds - Distrinet, KU Leuven

{firstname.lastname}@cs.kuleuven.be

## Abstract

Embedded domain specific languages (EDSLs) are used to represent special-purpose code in a general-purpose language and they are used for applications like vector calculations and run-time code generation. Often, code in an EDSL is compiled to a target (e.g. GPU languages, JVM bytecode, assembly, JavaScript) and needs to interface with other code that is available at that level but uses other data representations or calling conventions.

We present an approach for safely making available such APIs in a typed EDSL, guaranteeing correct conversions between data representations and the respect for calling conventions. When the code being interfaced with is the result of static compilation of host language code, we propose a way to auto-generate the needed boilerplate using meta-programming. We instantiate our technique with JavaScript as the target language, JS-Scala as the EDSL, Scala.js as the static compiler and Scala macros to generate the boilerplate, but our design is more generally applicable. We provide evidence of usefulness of our approach through a prototype implementation that we have applied in a non-trivial code base.

***Categories and Subject Descriptors*** D.2.12 [*Software Engineering*]: Interoperability

***Keywords*** embedded domain specific languages, Scala, meta-programming

## 1. Introduction

Software can grow big and complex making it difficult to maintain and extend. Managing this growing complexity can be done by hiding it behind a layer of abstraction. Embedded domain specific languages (EDSLs) are such an abstraction, they represent special-purpose code in a general-purpose language, such as parsers, vector calculations and run-time code generation. EDSLs can compile to platforms other than the one its host language compiles to. Combined with embedded code generators they often target platforms such as JavaScript, JVM bytecode, assembly and OpenCL. As depicted in Figure 1, an EDSL program often needs to interface with library code from other programming languages (PLs) available on the target platform. Even if the library code is compiled with different data representations by different tools.

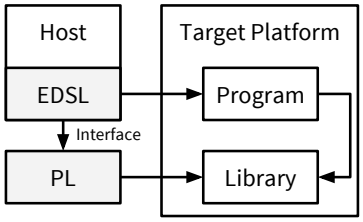


**Figure 1.** Interfacing on Target Platform

To eradicate mistakes while using EDSLs or PLs in general a type system can be used. Type systems ensure that no type errors can occur throughout the checked program's lifetime as long as its use abides by the type contract. When code type checks this property is proven for the specific code base. With type checked EDSLs and type checked libraries we have two safety properties, i.e., no type errors will occur in the EDSL program or the library. The problem is that interfacing between an EDSL and a library (as in Figure 1) can break these properties if incorrect representations of types are used. If our EDSL for example represents a function as a function on the target platform while our library represents a function as an object on the target platform it is obvious that type problems will occur. Common solutions to the problem require the EDSL user to provide the type information manually which implies another source of bugs and boilerplate.

In this document we instantiate the problem with JavaScript as the target platform, JS-Scala as the typed EDSL and Scala libraries compiled with Scala.js. JS-Scala (Kossakowski et al. 2012) is a typed EDSL for run-time JavaScript generation in Scala. It uses the Lightweight Modular Staging (LMS) framework by Rompf and Odersky (2010). Scala.js (Doeraene 2013) is a separate compiler back-end for Scala and statically compiles Scala code to JavaScript with its own representation of objects and classes.

The goal of the project is to enforce the library's and the EDSL's type contract to preserve the type-safety of both, even while they interface with each other. First, we provide a way of tracking the library's types in the type system of the EDSL. Next, we automatically generate boundary APIs for EDSLs to safely interface with the target library in their respective environment. The program (EDSL and library code) is made type-safe by reflecting the type constraints of the library in the generated EDSL APIs so that no ill-typed calls are possible. If we apply this to Figure 1 the interface would be type checked instead.

We describe our general solution in Section 2 and explain how we solve the issue of well-typedness and boilerplate by tracking the library's representations in the type system and by generating safe boundary APIs. Next, we highlight technical details regarding

the type tracking and generation in Section 3. We provide some evidence of usefulness by explaining our own use of the approach in a non-trivial code base in Section 4. Finally, we conclude with some future and related work in Sections 5 and 6.

## 2. Safe Boundary APIs

With the general description of our idea in mind, we now explain our solution more concretely by instantiating it with the previously mentioned technologies: JS-Scala and Scala.js.

To understand our approach, you need to understand how to read the types of the JS-Scala DSL. JS-Scala re-uses the Scala type system to make it a typed DSL for JavaScript. It inherits the technique from LMS which consists of annotating DSL types with a type constructor named `Rep`. A `String` type for example represents a regular Scala `String` while a `Rep[String]` type represents a JavaScript `String` that JS-Scala generates. `Rep[X]` adds information for the entire `X` type, it indicates that `X` in its entirety is a JavaScript `X` even if it itself is a compound type, e.g., `Rep[Array[Int]]` is a JavaScript array of JavaScript numbers.

***Tracking run-time representations*** The library code that we are targeting from JS-Scala is compiled with Scala.js. Scala.js is a Scala to JavaScript compiler that has its own representation of Scala classes and objects. JS-Scala is a JavaScript DSL in Scala and its aim is to closely represent a statically typed version of JavaScript itself. This means we have two value representations for one type. A simple example is the array; in Scala.js an array is a Scala object with its own object representation while a JS-Scala array is just a plain JavaScript array.

We track these representations in the types of our boundary APIs to make them safe. Table 1 shows the correspondence between Scala, JavaScript and our JS-Scala types. We tag types with `Sjs` to indicate that they are Scala.js representations in JS-Scala. Certain types have the same representation in Scala.js as in JS-Scala since they re-use standard JavaScript types, for example `String`. We will refer to such types as *primitives*.

Note that `Sjs[X]` does not necessarily imply something for the entire `X` type like `Rep[X]` does. In `Rep[Y]` we know that `Y` is entirely a JavaScript type but in `Sjs[X]` the only information that we learn from the type is that `X`'s outermost type constructor is a representation of a Scala.js type. So if `X` were to be `List[Z]` we know that it is a Scala.js list of JavaScript Zs. Unlike JS-Scala's `Rep` we can have mixed situations, for example, a Scala.js array of JavaScript arrays: `Rep[Sjs[Array[Array[Int]]]]`.

Complementing this representation of Scala.js types in JS-Scala, we provide a library to convert Scala.js representations to JS-Scala and vice-versa. Functions, for example, have different representations, a Scala.js function compiles to an object representation rather than a regular JavaScript function. Our utility library can convert between JS-Scala's (JavaScript) and Scala.js's function: `Rep[A ⇒ B]` ⇔ `Rep[Sjs[A ⇒ B]]`. We also provide conversions for sequences, maps, tuples and options.

***Generating boundary APIs*** In Figure 2, we show a graphical representation of a typical Scala.js/JS-Scala program. `Vec` is a regular Scala.js library and statically compiles to JavaScript (`Vec.Sjs`). `VecExtra` uses JS-Scala for run-time JavaScript code generation (`JS-VecExtra`). Our approach makes the functionality exposed in `Vec.Sjs` safely available for `VecExtra` by generating boundary APIs for JS-Scala. The APIs use the `Sjs` type representation to preserve type-safety across JS-Scala/Scala.js boundaries.

***Safe boundary APIs by example*** An example application of EDSLs is run-time code generation which has the benefit of being able to specialize code at run-time. In our example, we enhance a Scala vector library `Vec` by adding run-time generated functionality with JS-Scala in an extra library `VecExtra`.

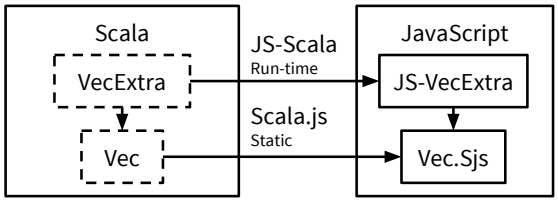


**Figure 2.** Compilation Scheme

```
// Vec
@JsScalaProxy trait Vec {
  @JSExport def scale(f: Int): Vec
  ...
}

// VecExtra
trait VecExtra extends Vec.VecLib {
  type VecE = Rep[Sjs[Vec]]
  def scaleOpt(n: Int, v: VecE): VecE =
    n match {
      case 1 => v
      case n => v.scale(lift(n))
    }
  ...
}
```

**Figure 3.** Vector DSL Excerpt

This small example is for didactic purposes. Run-time code generation libraries more typically target a low-level language like OpenCL or C instead of JavaScript, but the same issues arise in such a setting. Our reason for targeting JavaScript is explained in Section 4 where we talk about our own use of the prototype implementation.

Figure 3 displays the `Vec` and `JS-VecExtra` libraries. The main `Vec` library is just regular Scala code that defines a trait[1] `Vec` with a method to scale itself by an integer. The Scala.js `@JSExport` annotation on the method marks that it should be accessible from the compiled JavaScript code. Making this functionality accessible from JS-Scala is done by invoking our meta-program with the `@JsScalaProxy` annotation on the `Vec` trait. The meta-program creates a trait `Vec.VecLib` to store the JS-Scala boundary APIs.

The `VecExtra` library contains the run-time code generation in the JS-Scala DSL. It imports the boundary APIs by extending the `Vec.VecLib` trait, which makes the methods of `Vec` (such as `scale`) available for the JS-Scala DSL. The trait defines, for example, `Rep[Sjs[Vec]].scale(f : Rep[Int]) : Rep[Sjs[Vec]]`.

In `scaleOpt` we eliminate scaling by one by performing a check on the argument. If the argument is (statically known to be) one, the generated code does nothing and returns the vector immediately. Otherwise, it dispatches to the statically compiled scale method. This sort of toy optimization can easily expand into complex specialized programs while maintaining high-level abstractions. Note that there is no boilerplate required for the JS-Scala user to safely interface with statically compiled Scala.js code.

## 3. Transformation

The previous section roughly explained our solution for preserving type-safety in both JS-Scala programs and Scala.js libraries while allowing interaction between them. Now, we fill in some more details regarding the approach by focusing on both the `Sjs` tagging function and our actual generation. This section is necessarily Scala

[1] Traits can be thought of as Java interfaces allowing implemented methods.

| Scala.js type | JavaScript type | JS-Scala Interop type |
|---|---|---|
| String | string | Rep[String] |
| Boolean | boolean | Rep[Boolean] |
| Byte, Short, Int, Float, Double | number | Rep[Byte], ... |
| Unit | undefined | Rep[Unit] |
| Array[Int] | Scala.js Array of numbers | Rep[Sjs[Array[Int]]] |
| Array[js.Array[Int]] | Scala.js Array of JavaScript arrays of numbers | Rep[Sjs[Array[Array[Int]]]] |
| Array[Option[Int]] | Scala.js Array of Scala.js Options of numbers | Rep[Sjs[Array[Sjs[Option[Int]]]]] |
| UserDefined[Int] | Scala.js UserDefined type | Rep[Sjs[UserDefined[Int]]] |

**Table 1.** Type Correspondence

specific since we target the Scala type system and use Scala meta-programming tools (Scala macro annotations) to generate extra functionality at compile-time.

***Type transformation*** A type transformation applied to the type signatures of existing library code tracks the different representations. The transformation function $\mathcal{G}$, $\mathcal{T}$ and $\mathcal{H}$ are given in Figure 4 where $\mathcal{G}$ and $\mathcal{T}$ tag types with `Rep` and `Sjs` while $\mathcal{H}$ tags type definitions. The type definitions that we support are found in parameter lists of method headers (`def z[A <: AnyRef](x : A)`) or in existential type declarations (`t forSome { type t <: AnyRef }`). For existential type declarations there is a clause in $\mathcal{T}$ to invoke $\mathcal{H}$ and for method headers we invoke $\mathcal{H}$ manually.

For the Scala enthusiasts among you that noticed some type features being left out—we support the types given in $\mathcal{T}$ and omit more exotic types that are possible in Scala, such as structural types to keep the prototype simpler.

We apply $\mathcal{G}$ to all types in method headers, tagging them with `Rep` and invoking $\mathcal{T}$ on them. $\mathcal{T}$ recursively tags all types with `Sjs` except for bound type variables and primitives (types with the same representation in JS-Scala and Scala.js).

Bound type variables are type variables that are bound by having them as a parameter, e.g., in `def m[A](x : A)` the `A` is a bound type variable. We can ignore bound type variables because we do not support run-time reflection. This implies that method bodies using type parameters remain polymorphic to the type, polymorphic even in values with types from another language. This allows us to interface with polymorphic Scala.js code using any value, even those foreign to the Scala.js run-time while still preserving the type constraints of the compiled Scala.js library. A concrete example is filling Scala.js arrays with raw DOM elements, i.e., `Rep[Sjs[Array[HTMLElement]]]`.

***Generating boilerplate*** Our solution to the problem of combining JS-Scala and Scala.js code is the correct generation of JS-Scala boilerplate. We use Scala macro annotations, an experimental feature of the Scala macro paradise plugin[2]. We explain the gist of this transformation without focusing on technical details inherent to LMS. Full generation details can be found on our project page[3].

In Figure 5 we demonstrate a complete transformation of the original code (top) to a version where the JS-Scala interfacing boilerplate is added (bottom). We generate JS-Scala boilerplate for the `scale` method on `Vec`. The generated boilerplate is added in the `Vec` companion object[4]. JS-Scala uses traits as its modules and as such there is one trait added (`VecLib`) that serves as the module containing the safe boundary APIs for `Vec`.

[2] http://docs.scala-lang.org/overviews/macros/annotations.html

[3] https://github.com/Tzbob/scala-js-2-js-scala/releases/tag/gpce15

[4] For those unfamiliar with companion objects, think of them as a special kind of global object, like singletons that share the private access scope with their companion class or trait.

```
object Vec {...}

trait Vec[A] { @JSExport def scale(f: Int): Vec }
```

⇓

```
object Vec {
  ...
  trait VecLib extends DelegatorLib {
    trait VecOps {
      val self: Rep[Sjs[Vec]]

      def scale(n: Rep[Int]): Rep[Sjs[Vec]] =
        callDef(self, "scale", List(n))
    }

    implicit def addOps(x: Rep[Sjs[Vec]]): VecOps =
      new VecOps { val self = x }
  }
}

trait Vec[A] { def scale(f: Int): Vec }
```

**Figure 5.** Vector DSL transformation

`VecLib` consists of two parts: the generated JS-Scala boilerplate with the version of `scale` for `Rep[Sjs[Vec]]` and an implicit conversion to make this method available on `Rep[Sjs[Vec]]`.

## 4. Case Study: MT-FRP in Scala

We have evaluated our technique by applying our prototype to a generative programming research project. In our project we look at providing a platform for declarative development of interactive web applications. We approach this goal by extending the Functional Reactive Programming (FRP) paradigm to a client/server model in a multi-tier (or tierless) language. We refer to this as Multi-Tier Functional Reactive Programming (Reynders et al. 2014).

FRP is an abstraction for time-dependent values and is 'reactive' in the sense that changes to values propagate to their dependencies. An FRP programmer no longer has to manually synchronize change between dependencies.

A multi-tier language is a language that allows the programmer to write both client and server code in the same language and even within the same code base. We prototype a multi-tier language in Scala by mimicking its features with a combination of regular Scala and JS-Scala.

The client/server extension that we provide to FRP basically connects client and server FRP run-times to each other. The run-times need the same semantics which we preferably achieve without writing the library twice. We write it once in Scala and compile it for both the JVM and for JavaScript.

This introduces the problem described in Section 1, we want safe and boilerplate-free interaction from our typed multi-tier language (Scala + JS-Scala) to our pre-compiled library. We apply the technique described in Section 2 with the client FRP run-time as

$$\mathcal{G}[\![X]\!] \stackrel{\text{def}}{=} \texttt{Rep}[\mathcal{T}[\![X]\!]] \qquad \mathcal{T}[\![X[I_1, ..., I_n]]\!] \stackrel{\text{def}}{=} \texttt{Sjs}[X[\mathcal{T}[\![I_1]\!], ..., \mathcal{T}[\![I_n]\!]]]$$

$$\mathcal{T}[\![A \text{ with } B]\!] \stackrel{\text{def}}{=} \mathcal{T}[\![A]\!] \text{ with } \mathcal{T}[\![B]\!] \qquad \mathcal{T}[\![X \text{ forSome \{ Defs \}}]\!] \stackrel{\text{def}}{=} \mathcal{T}[\![X]\!] \text{ forSome \{ Defs.map}(\mathcal{H}) \text{ \}}$$

$$\mathcal{T}[\![X\#Y]\!] \stackrel{\text{def}}{=} \texttt{Sjs}[X\#Y] \qquad \mathcal{T}[\![X.type]\!] \stackrel{\text{def}}{=} \texttt{Sjs}[X.type]$$

$$\mathcal{T}[\![X]\!] \stackrel{\text{def}}{=} \begin{cases} X & \text{if } Primitive(X) \vee BoundVariable(X) \\ \texttt{Sjs}[X] & \text{otherwise} \end{cases}$$

$$\mathcal{H}[\![X <: P_1 <: ... <: P_n >: S_1 >: ... >: S_m]\!] \stackrel{\text{def}}{=} X <: \mathcal{T}[\![P_1]\!] <: ... <: \mathcal{T}[\![P_n]\!] >: \mathcal{T}[\![S_1]\!] >: ... >: \mathcal{T}[\![S_m]\!]$$

**Figure 4.** Type Transformation

the pre-compiled target library and our multi-tier code (which uses JS-Scala) as the typed EDSL.

We generate safe boundaries in our 1k line run-time[5] for 48 interfaces and make use of them in a 1.5k line framework[6] without issues, even with advanced Scala features like existential types.

The use of boundary API generation has been an improvement to the project. Originally we manually wrote wrappers which duplicated public interfaces (JS and JVM) to the run-time library and added lots of JS-Scala boilerplate. Adding or changing features meant editing three different places which was error-prone. On the downside, the generated boundary APIs can have very cluttered types from the `Sjs` tagging. In our use-case this was not that big of a problem since the boundary APIs were always used in the internals of the framework.

## 5. Future Work

In future work we would like to make our transformation more generally applicable. In the introduction we already claimed that we can apply our approach to different platforms yet our prototype implementation does not reflect this. We identify two main directions in which we can generalize our prototype, by accepting different library languages (not just Scala) and by allowing different LMS back-ends other than JavaScript (JS-Scala).

***Library languages*** At the moment we only accept one library language which is Scala itself but the only requirement is a correct type conversion towards the EDSL. The macros that gather type information from Scala itself can get the type information from a different source, e.g., TypeScript type definitions or C header files.

For JS-Scala projects in particular this is a good addition. Many typed languages compile to JavaScript but Scala.js is arguably one of the less popular ones when compared to GWT by Google (2006) or TypeScript by Microsoft (2012). Being able to safely interface with TypeScript's typed APIs gives JS-Scala programmers easy access to popular JavaScript frameworks such as jQuery.

***Target platforms*** At the moment we only support one target platform—JavaScript. LMS allows much more and has back-ends for other platforms like C and Scala.

With small adjustments we can apply our approach to LMS itself and enable all its back-ends. Combined with the generalization to accept different libraries this for example leads to a safe and convenient way of interfacing with C code.

## 6. Related Work

To the best of our knowledge there is no directly related work that discusses boundaries between typed EDSLs and typed languages. If we widen our view to all FFIs for high-level languages it is impossible to do the work justice in the little amount of room we have. Instead, we focus on work that solves the same goals (safe & boilerplate-free) using similar methods (embedding types).

C $\Rightarrow$ HS by Chakravarty (2000) is an approach to access C libraries from Haskell. Similar to our approach they do not require an extra interface description language. They gain the required type information directly from the C header files of the library and also provide a Haskell library to convert from Haskell to C representations and vice-versa.

Blume (2001) presents a FFI for SML/NJ that is based on the ability of ML programs to inspect and manipulate C data structures directly. They have a strong focus on embedding the C type system within ML types and apart from variable argument functions they succeed in a complete embedding. Using this (nearly) exact representation of the C type system they generate SML interface APIs from C header files similar to our approach.

Contrary to our approach the other work does not re-use existing types for C representations but instead have C specific types, e.g., `CInt` instead of `Sjs[Int]`.

## Acknowledgments

This research is partially funded by the Research Fund KU Leuven. Dominique Devriese holds a postdoctoral fellowship of the Research Foundation - Flanders (FWO).

## References

M. Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". *ENTCS*, pages 36–52, 2001.

M. M. T. Chakravarty. C $\rightarrow$ HASKELL, or Yet Another Interfacing Tool. In *IFL*, pages 131–148. Springer-Verlag, 2000.

S. Doeraene. Scala.js: Type-Directed Interoperability with Dynamically Typed Languages. Technical Report EPFL-REPORT-190834, EPFL, 2013.

Google. Google web toolkit. online, 2006. URL http://gwtproject.org/.

G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript as an embedded DSL. In *ECOOP*, pages 409–434. Springer-Verlag, 2012.

Microsoft. Typescript. online, 2012. URL http://www.typescriptlang.org/.

B. Reynders, D. Devriese, and F. Piessens. Multi-Tier Functional Reactive Programming for the Web. In *Onward!*, pages 55–68. ACM, 2014.

T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010.

[5] https://github.com/Tzbob/hokko

[6] https://github.com/Tzbob/s-mt-frp